PYTHON FOR DATA SCIENCE & AI

Dasun Athukoralage
web: www.dasuna.me
email: dasun@nirvanaclouds.com

Python Libraries and Data Processing

What is a Python Library?

In Python, libraries are collections of **pre-written code and modules** that provide a wide range of functionality to simplify and expedite the development of software.

These libraries contain **functions, classes, and methods** that can be imported and used in your Python programs, reducing the need to reinvent the wheel for common tasks.

Module vs Library

In Python, a **module** is **a file with the .py extension** that contains Python code. It can contain functions, classes, variables, and other Python objects. Modules are used to organize and reuse code.

A **library** is a **collection of related modules**. It can be a **standard (built-in) library**, which comes pre-installed with Python, or a **third-party library**, which can be installed using a package manager such as **pip**.

Standard Library

Python comes with a comprehensive standard library that includes modules for tasks such as file I/O, regular expressions, networking, and more.

These modules are included with every Python installation and can be used without additional installations.

Standard Library

You can find the complete Python Standard Library documentation here:

https://docs.python.org/3/library/index.html

Open Source Libraries

Open-source libraries are collections of pre-written, reusable code components that are made available to the public under an open-source license.

These libraries are developed collaboratively by a community of programmers, and they provide specific functionalities or solve common problems in software development.

Open Source Libraries

Open-source libraries are a fundamental part of the software development ecosystem and play a crucial role in various aspects of the field.

Benefits of using open-source libraries: time-saving, code reusability, and community support.

Selecting relevant Libraries

Selecting the right library for a software development task is of paramount importance for a variety of reasons.

The choice of library can significantly impact the project's success, efficiency, maintainability, and overall development process.

Selecting relevant Libraries

Common open-source libraries for various domains

Ex:

- **NumPy** for numerical computing
- **Pandas** for data analysis
- **Matplotlib** for data visualization

NumPy Library

NumPy Library

NumPy, which stands for "**Numerical Python**," is a fundamental open-source library in Python that provides support for working with **large, multi-dimensional arrays and matrices**, along with a collection of mathematical functions to operate on these arrays.

It is a cornerstone library for **data manipulation, analysis**, and **scientific computing** in Python.

NumPy Library

Before using NumPy, make sure it's installed. You can install it using pip if you don't already have it:

pip install numpy (or pip3 install numpy)

NumPy User Guide:

https://numpy.org/doc/stable/user/index.html

NumPy Library

Why NumPy?

- In Python we have **lists** that serve the purpose of arrays, but they **are slow to process**.
- NumPy aims to provide an array object that is up to **50x faster than traditional Python lists**.

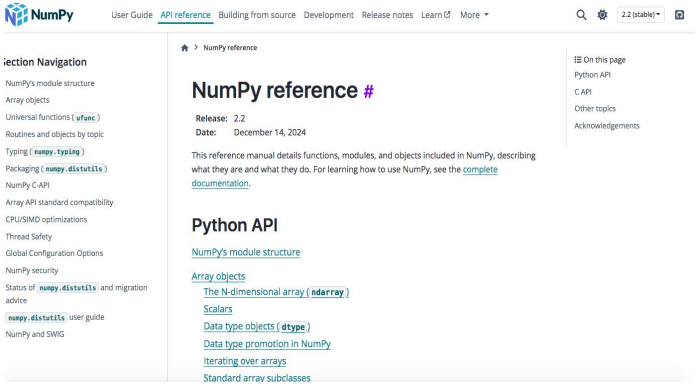
Where is the NumPy Codebase?

https://github.com/numpy/numpy

NumPy Library

NumPy Reference

- https://numpy.org/doc/stable/reference/index.html
- This reference manual details functions, modules, and objects included in NumPy, describing what they are and what they do.



NumPy Library

Array creation:

You can create an array from a regular Python list or tuple using the array function.

```
import numpy as np


a = np.array([2, 3, 4])


print(a)
print(type(a))      # <class 'numpy.ndarray'>
```

NumPy Library

```
import numpy as np

# Creating NumPy Arrays
array1 = np.array([1, 2, 3, 4, 5])
array2 = np.array([6, 7, 8, 9, 10])

# Performing Basic Operations
# Addition
result_addition = array1 + array2
```

NumPy Library

```
# Subtraction
result_subtraction = array2 - array1

# Multiplication
result_multiplication = array1 * array2

# Division
result_division = array2 / array1
```

NumPy Library

```
# Square Root
result_sqrt = np.sqrt(array1)

# Summation

result_sum = np.sum(array1)

# Mean (Average)

result_mean = np.mean(array2)
```

NumPy Library

```
# Maximum and Minimum

result_max = np.max(array1)

result_min = np.min(array2)
```

NumPy Library

Multidimensional Arrays

```
array_1 = np.array([[1, 2, 3], [4, 5, 6]])   # 2-D array
#An array that has 1-D arrays as its elements is called a
2-D array.


array_2 = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
# 3-D array
#An array that has 2-D arrays (matrices) as its elements is
called 3-D array.
```

NumPy Library

Some useful array attributes..

Shape

The shape attribute of a NumPy array describes its dimensions. It returns a tuple representing the size of the array in each dimension.

```
array = np.array([[1, 2, 3], [4, 5, 6]])
print(array.shape)  # Output: (2, 3)
```

NumPy Library

Some useful array attributes..

dtype

The dtype attribute specifies the data type of the elements stored in the NumPy array, such as int32, float64, or bool.

```
array = np.array([1, 2, 3])
print(array.dtype)  # Output: int64
```

NumPy Library

Some useful array attributes..

dtype

Note: On most modern 64-bit systems, NumPy defaults to `int64` for integers. However, this could vary depending on the system architecture. For 32-bit systems, the output might be `int32`.

NumPy Library

Some useful array attributes..

dtype

```
array = np.array([1, 2, 3], dtype=np.float64)
print(array.dtype)  # Output: float64
```

NumPy Library

Some useful array attributes..

size

The `size` attribute returns the total number of elements in the array.

```
array = np.array([[1, 2], [3, 4], [5, 6]])
print(array.size)  # Output: 6
```

NumPy Library

Some useful array attributes..

ndim

The ndim attribute indicates the number of dimensions (axes) in the array.

```
array = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(array.ndim)  # Output: 3
```

NumPy Library

NumPy Array Indexing

```
array_1 = np.array([1, 2, 3, 4])
print(array_1[2]) # output is 3


array_2 = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st row: ', array_2[0, 1])
```

NumPy Library

NumPy Array Indexing

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print(arr[0, 1, 2])  # Output ????
```

NumPy Library

NumPy Array Indexing (Negative)

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('Last element from 2nd dim: ', arr[-2, -1])#output ???
```

NumPy Library

Boolean Indexing in NumPy

Boolean indexing allows us to **filter elements of an array based on a condition**. This is one of the most powerful features in NumPy for data manipulation, as it enables the **selection of specific elements without the need for loops**.

NumPy Library

Boolean Indexing in NumPy

How it Works:

1. **Boolean Array:** A condition applied to a NumPy array results in a boolean array, where each element is either `True` (condition met) or `False` (condition not met).
2. **Indexing with Boolean Array:** When a boolean array is used to index a NumPy array, only the elements corresponding to `True` are selected.

NumPy Library

Boolean Indexing in NumPy

```
import numpy as np

# Create a NumPy array
arr = np.array([10, 20, 30, 40, 50])

# Apply a condition (e.g., values greater than 25)
condition = arr > 25
```

NumPy Library

Boolean Indexing in NumPy

```
# Output the boolean array
print(condition)  # [False False  True  True  True]

# Use boolean indexing to filter values
filtered = arr[condition]
print(filtered)  # [30 40 50]
```

NumPy Library

Boolean Indexing in NumPy

* **Filtering Using Conditions**

You can directly filter elements of a NumPy array by applying conditions within the indexing brackets. This avoids the need to explicitly create a separate boolean array.

NumPy Library

Boolean Indexing in NumPy

* **Filtering Using Conditions**

```
# Create a NumPy array
arr = np.array([[50, 20, 60], [30, 40, 10]])

# Directly filter values greater than 25
filtered_direct = arr[arr > 25]
print(filtered_direct)  # [50 60 30 40]
```

NumPy Library

Boolean Indexing in NumPy

* **Combining Multiple Conditions**

You can combine multiple conditions using logical operators like & (and), | (or), and ~ (not). **Note that parentheses are necessary due to operator precedence**.

NumPy Library

Boolean Indexing in NumPy

* **Combining Multiple Conditions**

```
# Create a NumPy array
arr = np.array([10, 20, 30, 40, 50])

# Values greater than 20 and less than 50
filtered_combined = arr[(arr > 20) & (arr < 50)]
print(filtered_combined)  # [30 40]
```

NumPy Library

Array Initialization - numpy.zeros()

Syntax: `np.zeros(shape, dtype=float, order='C')`

`np.zeros()` is used to **create a new array filled with zeros**. It is particularly useful for initializing arrays when you know their size but want all the elements to start with zero.

NumPy Library

Array Initialization - numpy.zeros()

Syntax: `np.zeros(shape, dtype=float)`

Parameters:

shape (required)- Defines the dimensions of the array.
- Can be an integer (for a 1D array) or a tuple of integers (for multi-dimensional arrays).

dtype (optional) - Specifies the data type of the array's elements, such as `int`, `float`, or `complex`.

NumPy Library

Array Initialization - numpy.zeros()

```
Creating a 1D array of zeros:
arr = np.zeros(5)

Creating a 2D array of zeros:
arr = np.zeros((3, 4))
# Output:
# [[0. 0. 0. 0.]
#  [0. 0. 0. 0.]
#  [0. 0. 0. 0.]]
```

NumPy Library

Array Initialization - numpy.full()

Syntax: **np.full(shape, fill_value)**

Return a new array of given shape and type, filled with *fill_value*.

NumPy Library

Array Initialization - numpy.full()

Syntax: **np.full(shape, fill_value)**

Parameters:

shape (required)- Defines the dimensions of the array.

fill_value (required) - The constant value used to fill the array. ***scalar or array_like.***

NumPy Library

Array Initialization - numpy.full()

```
#Create a 2D array filled with 7
arr = np.full((2, 3), 7)

#Create a 2D array filled with array [1, 2]
arr = np.full((2, 2), [1, 2])
```

NumPy Library

Array Initialization - numpy.empty()

Syntax: **np.empty**(shape, **dtype**=**float**)

Return a new array of given shape and type, without initializing entries.

This means the array will contain arbitrary values (essentially, **uninitialized garbage values**).

NumPy Library

Array Initialization - `numpy.empty()`

Why Use `numpy.empty()`?

1. **Performance**: If you know you'll overwrite all elements of the array shortly after creation, using `numpy.empty()` avoids the overhead of initialization, making it faster.
2. **Flexibility**: Sometimes, you might not need the array elements to be initialized upfront.

NumPy Library

Array Initialization - numpy.empty()

```
import numpy as np

arr = np.empty((2, 2))
print(arr)
```

Pandas Library

Pandas Library

Pandas is a popular open-source data manipulation and data analysis library for the Python programming language.

It is widely used in **data science, data analysis, and machine learning** for tasks involving structured data.

Pandas Library

Before using Pandas, make sure it's installed. You can install it using pip if you don't already have it:

pip install pandas (or pip3 install pandas)

Pandas User Guide:

https://pandas.pydata.org/docs/user_guide/index.html

Pandas Library

What is DataFrame in Pandas?

A **DataFrame** in Pandas is like a table in Excel or a database: it's a **two-dimensional structure with rows and columns**.

Each column can have a different data type (e.g., numbers, text, dates).

Pandas Library

Example 1: Creating a DataFrame

```
import pandas as pd

# Creating a DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 40]}
df = pd.DataFrame(data)
print(df)
```

Pandas Library

Example 1: Creating a DataFrame

Above code creates a Pandas DataFrame from a dictionary and prints it:

	Name	Age
0	Alice	25
1	Bob	30
2	Charlie	35
3	David	40

Pandas Library

Example 1: Accessing cells in a DataFrame

1. Using loc (label-based):

`df.loc[row_label, column_label]`

Ex: `print(df.loc[1, 'Name'])` # Output: Bob

2. Using iloc (index-based):

`df.iloc[row_index, column_index]`

Ex: `print(df.iloc[1, 0])` # Output: Bob

Pandas Library

Example 1: Locate Row

```
#refer to the row index:

print(df.loc[0])
```

Output:
```
Name  Alice
Age      25
Name: 0, dtype: object
```

Pandas Library

Example 1: Return Multiple Rows

```
#use a list of indexes:

print(df.loc[[0, 1]])
```

Output:

```
   Name  Age
0  Alice   25
1    Bob   30
```

Pandas Library

Example: Named Indexes

Add a list of names to give each row a name:

```
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

df = pd.DataFrame(data, index = ["day1", "day2", "day3"])

print(df)
```

Pandas Library

Example: Named Indexes

	calories	duration
day1	420	50
day2	380	40
day3	390	45

```
#refer to the named index:

print(df.loc["day2"])

print(df.loc[["day1","day2"]])  # return two rows
```

Pandas Library

___

Example: Read Columns

df['column_name']

Ex:

```
#Accessing the 'Age' column

age_column = df['Age'] # Using brackets

print(age_column)
```

Pandas Library

Useful Attributes: DataFrame.shape

- **Description**: Returns the dimensions of the DataFrame as a tuple (rows, columns).

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
print(df.shape)  # Output: (2, 2)
```

Pandas Library

Useful Attributes: DataFrame.columns

- **Description**: Returns an **Index object** containing column names of the DataFrame.

```
df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
print(df.columns)  # Output: Index(['A', 'B'], dtype='object')
print(list(df.columns))  # Convert to a list: ['A', 'B']
```

Pandas Library

Useful Attributes: DataFrame.dtypes

- **Description**: Displays the **data types** of each column in the DataFrame.

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
print(df.dtypes)
# Output:
# A    int64
# B    int64
# dtype: object
```

Pandas Library

Useful Attributes: DataFrame.size

- **Description**: Returns the total number of elements in the DataFrame (rows × columns).

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
print(df.size)  # Output: 4 (2 rows × 2 columns)
```

Pandas Library

Useful Attributes: DataFrame.values

- **Description**: Returns the data as a **NumPy array**.

```
import pandas as pd
df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
print(df.values)
# Output:
# [[1 3]
#  [2 4]]
```

Pandas Library

Useful Functions: DataFrame.len()

- **Description**: The `len()` function is used to return the number of rows in the DataFrame.

```
data = {
   'Age': [25, 30, 35, 40],
   'Salary': [50000, 60000, 70000, 80000]
   }
df = pd.DataFrame(data)
print(len(df)) #output: 4
```

Pandas Library

Useful Functions: DataFrame.mean()

- **Description**: Calculates the **mean (average)** of **numerical columns** (By default, excludes NaN values.)

```
data = {
   'Age': [25, 30, 35, 40],
   'Salary': [50000, 60000, 70000, 80000]
   }
df = pd.DataFrame(data)
print(df.mean())
```

Pandas Library

Useful Functions: DataFrame.mean()

- **Description**: Calculates the **mean (average)** of **numerical columns** (By default, excludes NaN values.)

Output:

```
Age       32.5
Salary  65000.0
dtype: float64
```

Pandas Library

Useful Functions: DataFrame.sum()

- **Description**: Returns the **sum** of values for each **numerical column** (By default, excludes NaN values).

```
data = {
   'Age': [25, 30, 35, 40],
   'Salary': [50000, 60000, 70000, 80000]
   }
df = pd.DataFrame(data)
print(df.sum())
```

Pandas Library

Useful Functions: DataFrame.sum()

- **Description:** Returns the **sum** of values for each **numerical column** (By default, excludes NaN values).

Output:

```
Age           130
Salary     260000
dtype: int64
```

Pandas Library

Useful Functions: DataFrame.describe()

- **Description**: Provides a summary of statistics for numerical columns by default.Includes **count, mean, standard deviation, minimum, quartiles (25%, 50%, 75%), and maximum** values.

Pandas Library

Useful Functions: DataFrame.describe()

```
data = {
    'Age': [25, 30, 35, 40],
    'Salary': [50000, 60000, 70000, 80000]
    }
df = pd.DataFrame(data)
print(df.describe())
```

Pandas Library

Useful Functions: DataFrame.describe()

Output:

```
            Age         Salary
count   4.000000      4.000000
mean   32.500000  65000.000000
std     6.454972  12909.944487
min    25.000000  50000.000000
25%    28.750000  57500.000000
50%    32.500000  65000.000000
75%    36.250000  72500.000000
max    40.000000  80000.000000
```

Pandas Library

Filtering Rows in a DataFrame

```
import pandas as pd
# Example DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 27, 22, 32],
    'Score': [85, 70, 90, 88]
}
df = pd.DataFrame(data)
print(df)
```

Pandas Library

Filtering Rows in a DataFrame

```
# Filtering rows where Age > 25
filtered_df = df[df['Age'] > 25]


print(filtered_df)
```

Output:

	Name	Age	Score
1	Bob	27	70
3	David	32	88

Pandas Library

Data Manipulation

```
# Filtering data

young_people = df[df['Age'] < 35]

# Sorting data

sorted_df = df.sort_values(by='Age')

# Adding a new column

df['Salary'] = [50000, 60000, 70000, 80000]
```

Pandas Library

Data Manipulation

```
# Adding a new row

# Example DataFrame
df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})

# Adding a new row with `loc`
df.loc[len(df)] = [5, 6]

print(df)
```

Pandas Library

Data Manipulation

 # Dropping a column/row

Syntax: `DataFrame.drop(labels, axis=1, inplace=False)`

labels:

- The name(s) of the column(s) you want to drop. You can pass a single column name as a string or multiple column names as a list of strings.

Pandas Library

Data Manipulation

 # Dropping a column/row

Syntax: `DataFrame.drop(labels, axis=1, inplace=False)`

axis:

- Set `axis=1` to indicate that you want to drop columns. (For rows, you'd use `axis=0`.)

Pandas Library

Data Manipulation

 # Dropping a column/row

Syntax: **DataFrame.drop(labels, axis=1, inplace=False)**

inplace:

- If inplace=True, the DataFrame is modified directly, and no new object is returned.
- If inplace=False (default), a new DataFrame with the column(s) dropped is returned, and the original DataFrame remains unchanged.

Pandas Library

Data Manipulation

```
# Dropping a column/row

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Gender': ['F', 'M', 'M']
}
df = pd.DataFrame(data)
```

Pandas Library

Data Manipulation

```
# Dropping a column/row

df_dropped = df.drop('Age', axis=1)
print(df_dropped)
# Output
      Name Gender
0    Alice      F
1      Bob      M
2  Charlie      M
```

Pandas Library

Read CSV Files

```
import pandas as pd

df = pd.read_csv('data.csv')
print(df)
```

Pandas Library

Read JSON Files

```
import pandas as pd

df = pd.read_json('data.json')
print(df)
```

Matplotlib Library

Matplotlib Library

Matplotlib is a widely used open-source **data visualization library** in Python, and it serves a critical role in the field of data science, scientific research, and any application that involves data visualization.

Matplotlib Library

Before using Matplotlib, make sure it's installed. You can install it using pip if you don't already have it:

pip install matplotlib (or pip3 install matplotlib)

Matplotlib User Guide:

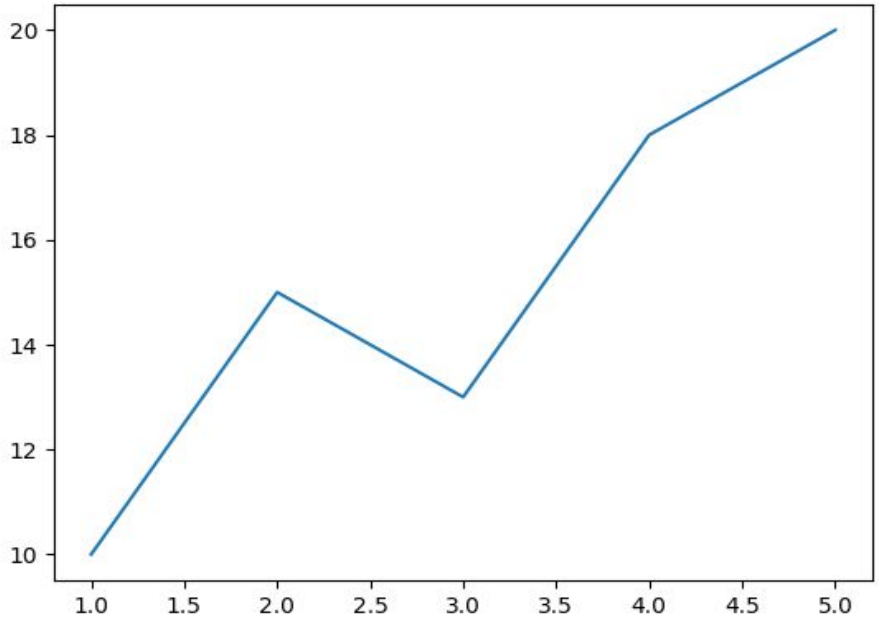
https://matplotlib.org/stable/users/index

Matplotlib Library

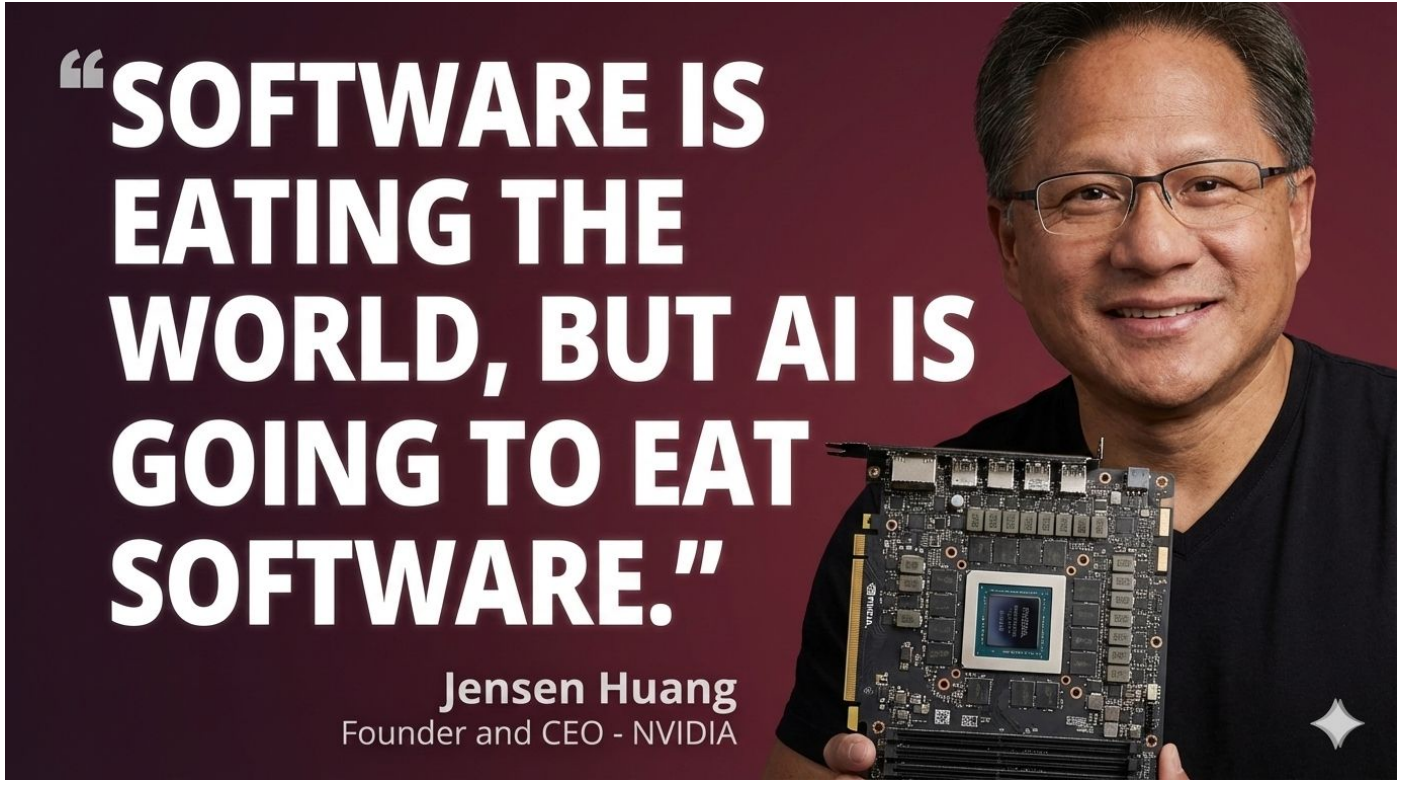
```
import matplotlib.pyplot as plt

# Sample data

x = [1, 2, 3, 4, 5]

y = [10, 15, 13, 18, 20]

# Create a line plot

plt.plot(x, y)

# Display the plot

plt.show()
```

Matplotlib Library

Output:



Thank You...!


“SOFTWARE IS EATING THE WORLD, BUT AI IS GOING TO EAT SOFTWARE.”
Jensen Huang
Founder and CEO - NVIDIA